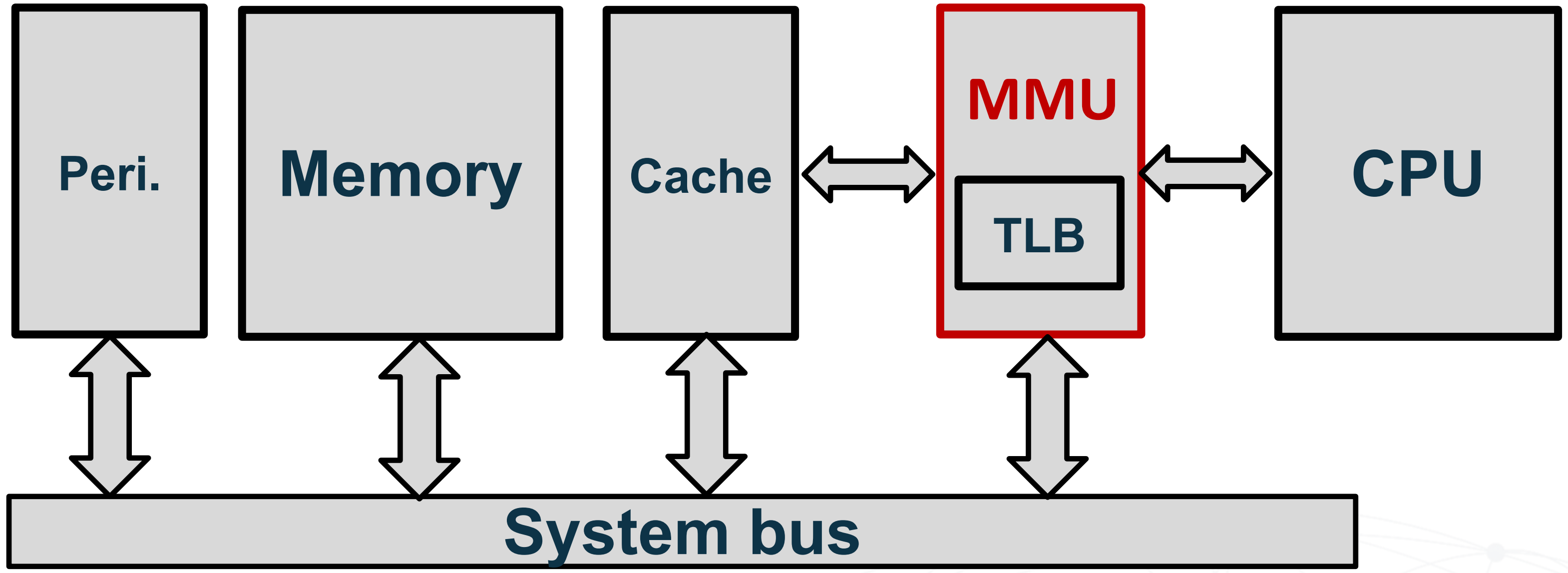
Understanding MPU Usage in Microcontroller-based Systems in the Wild

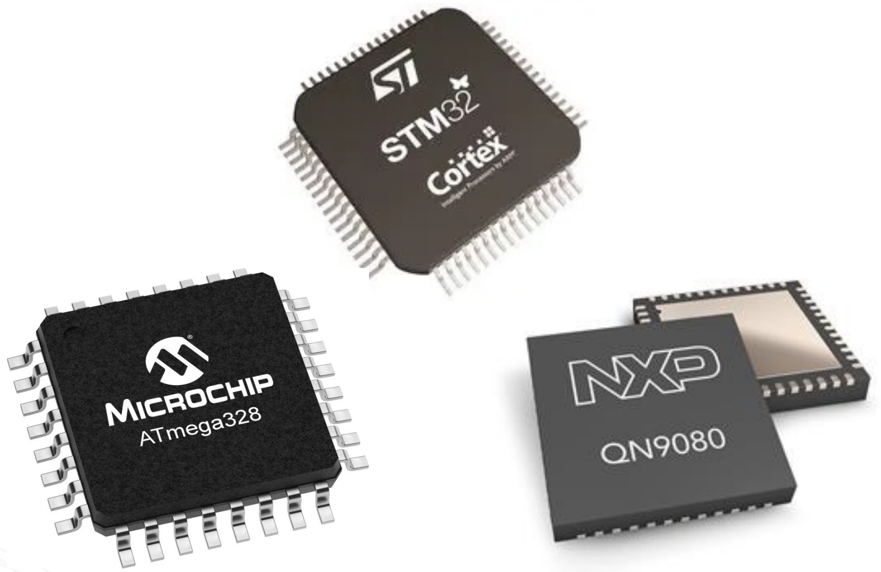
Wei Zhou, Zhouqi Jiang, Le Guan

Agenda

- **Introduction to Memory Protection Unit (MPU)**
- MPU adoption in the wild
- Common pitfalls and limitations in using MPU
- Mitigation suggestions
- Summary and disclosure

MMU on PC → MPU on MCU

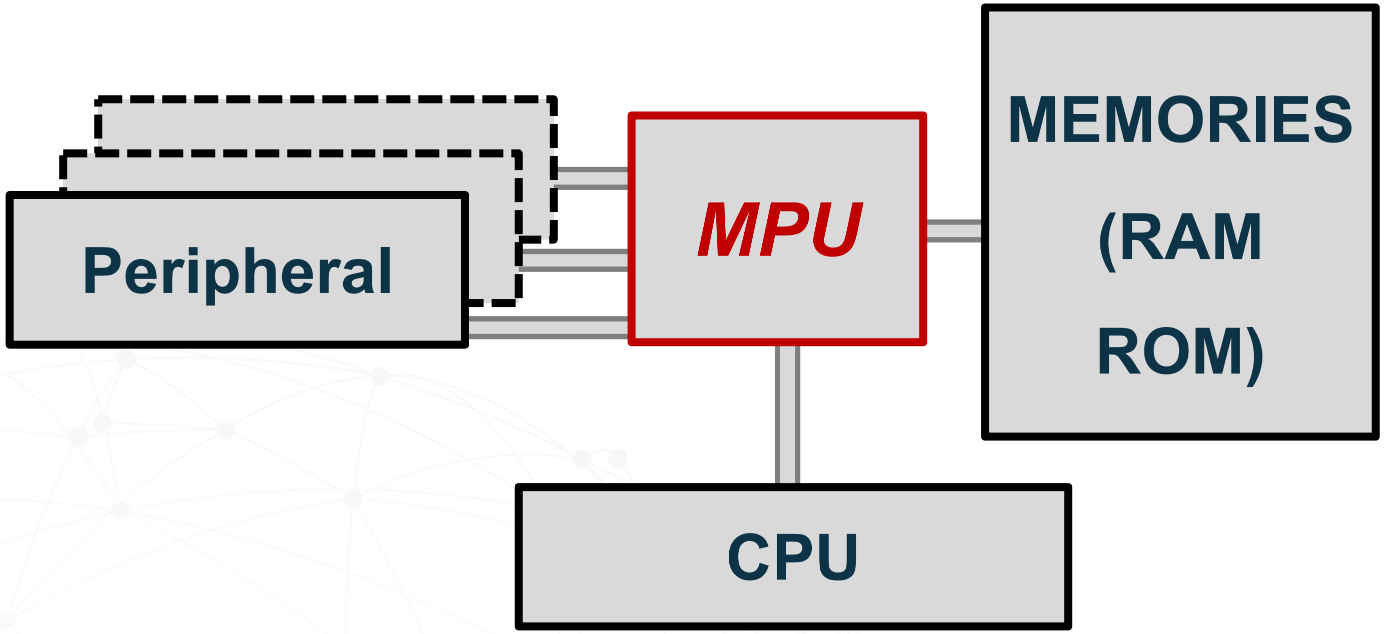




- The Memory Management Unit (MMU), a standard feature on traditional computing platforms, enforces page-based permission control **(R/W/X)**.
- MMU is absent in resource-restricted microcontroller units (MCUs) .
- As a stripped-down version of MMU, the **Memory Protection Unit (MPU)** provides basic security functions for MCUs, e.g., Arm Cortex-M series MCUs.

What is Memory Protection Unit (MPU)

- How MPU works?
  - For **a limited number** of configurable memory regions, MPU assigns access permissions (R/W/X) based on the current execution privilege level
  - A fault happens when a memory access violates the access permission
  - MPU can only be configured by privileged code



How to Program MPUs (PMSAv7)?

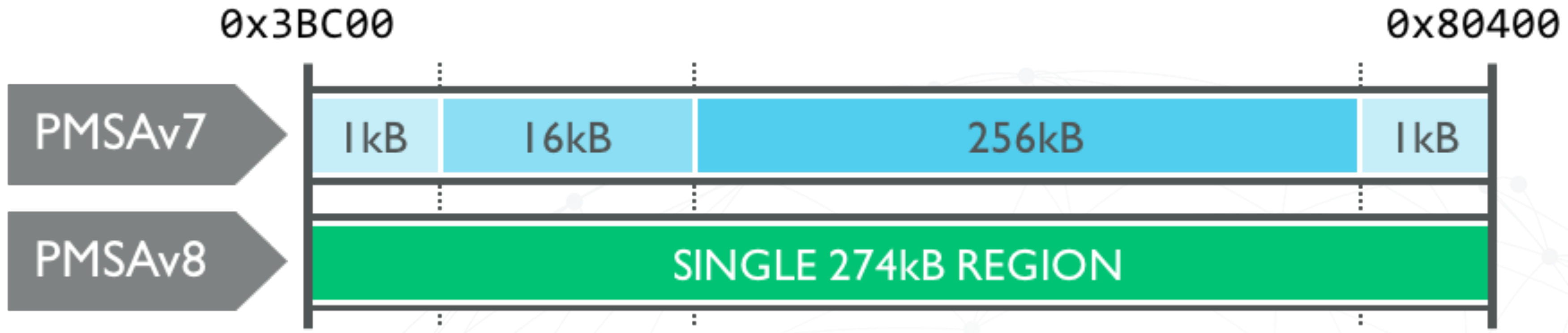
- **MPU Control Register (MPU_CTRL)**
  - **Enable bit:** enable the MPU
  - **The PRIVDEFENA bit:** enable the default memory map for privileged access
- **MPU Region Base Address Register (MPU_RBAR)**
  - Set the base address and the identifier of a memory region
- **MPU Region Attribute/Size Register (RASR)**
  - **AP and XN bit:** Set permissions (RWX) of a memory region in privileged and unprivileged level
  - **TEX, C and B bits:** Set the Attributes (e.g., cacheability and shareability) of a memory region
  - **Size :** Set the size of a memory region
- **Constrains on MPU memory region**

  (1) At least 32 bytes (2) Power of two

  (3) Must be aligned with 32 bytes (4) **Limited region numbers (M0+/3/4 up to 8 and M7 up to 16)**

What's new in MPUS (PMSAv8)?

- More MPU regions (up to 16 regions for both normal and secure world in M23 and M33)
- Use Start and Limit (end) address via separated MPU registers to define **any size of memory regions**, but still must be **32-byte aligned**



- PMSAv8 also introduces a new memory attribute indirection register (MPU_MAIR), making it easier for **multiple regions to share the same attribute**, while at the same time maintaining their own access permissions.

Agenda

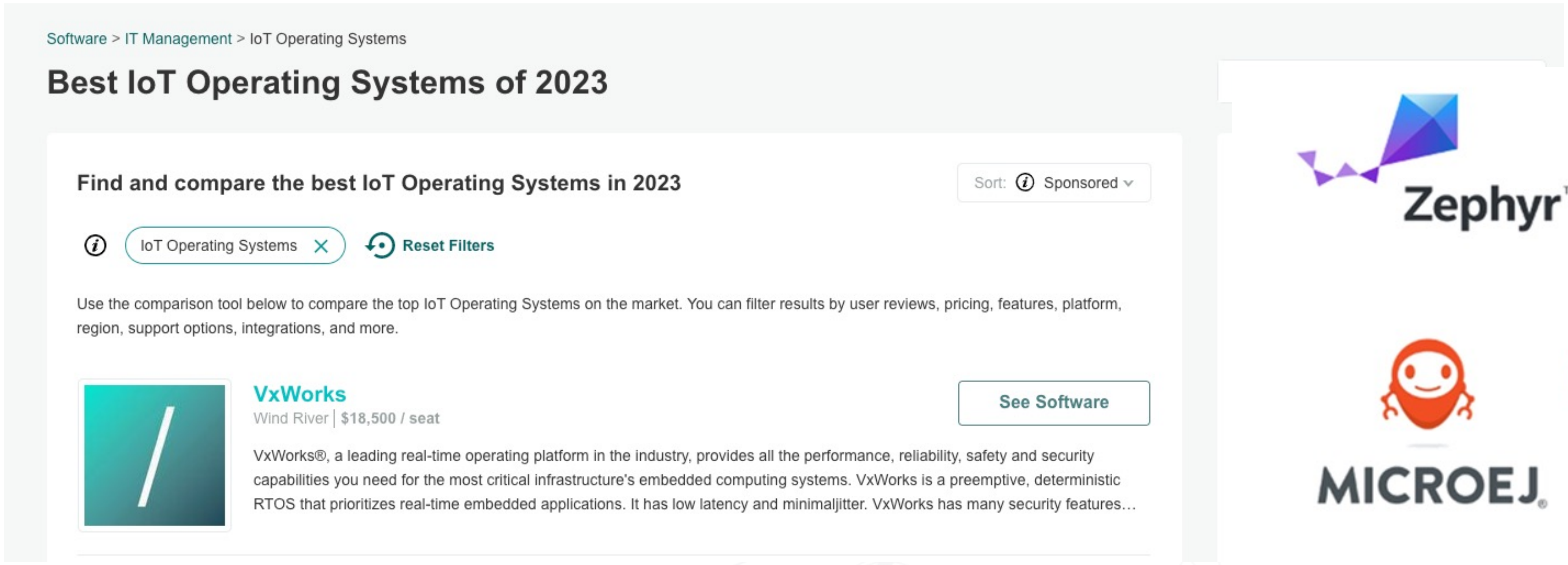
華中科技大學

- Introduction to Memory Protection Unit (MPU)
- **MPU adoption in the wild**
- Common pitfalls and limitations in using MPU
- Mitigation suggestions
- Summary and disclosure

MPU-enabled common security functions

- **Code Integrity Protection (CIP):** Code regions can be set as non-writable to prevent code injection and manipulation.
- **Data Execution Prevention (DEP):** Data regions like stack or heap can be set non-executable
- **Stack Guard (SG):** A redzone can be placed at the task stack boundary to detect stack overflows.
- **Kernel Memory Isolation (KMI):** User mode (unprivileged) code cannot access any memory belonging to the kernel space without invoking system calls
- **User Task Memory Isolation (TMI):** User mode (unprivileged) tasks can only access its own memory except explicitly shared memory regions that belong to other tasks or kernel.
- **Peripherals Isolation (PI):** Peripheral access is restricted to tasks having the privilege to access it.

MPU adoption in the wild



Software > IT Management > IoT Operating Systems

Best IoT Operating Systems of 2023

Find and compare the best IoT Operating Systems in 2023

Sort: Sponsored

IoT Operating Systems | Reset Filters

Use the comparison tool below to compare the top IoT Operating Systems on the market. You can filter results by user reviews, pricing, features, platform, region, support options, integrations, and more.

VxWorks
Wind River | $18,500 / seat

See Software

VxWorks®, a leading real-time operating platform in the industry, provides all the performance, reliability, safety and security capabilities you need for the most critical infrastructure's embedded computing systems. VxWorks is a preemptive, deterministic RTOS that prioritizes real-time embedded applications. It has low latency and minimaljitter. VxWorks has many security features…

We did first comprehensive study on MPU usage of top 30 IoT operating systems according to market rating.

Observation

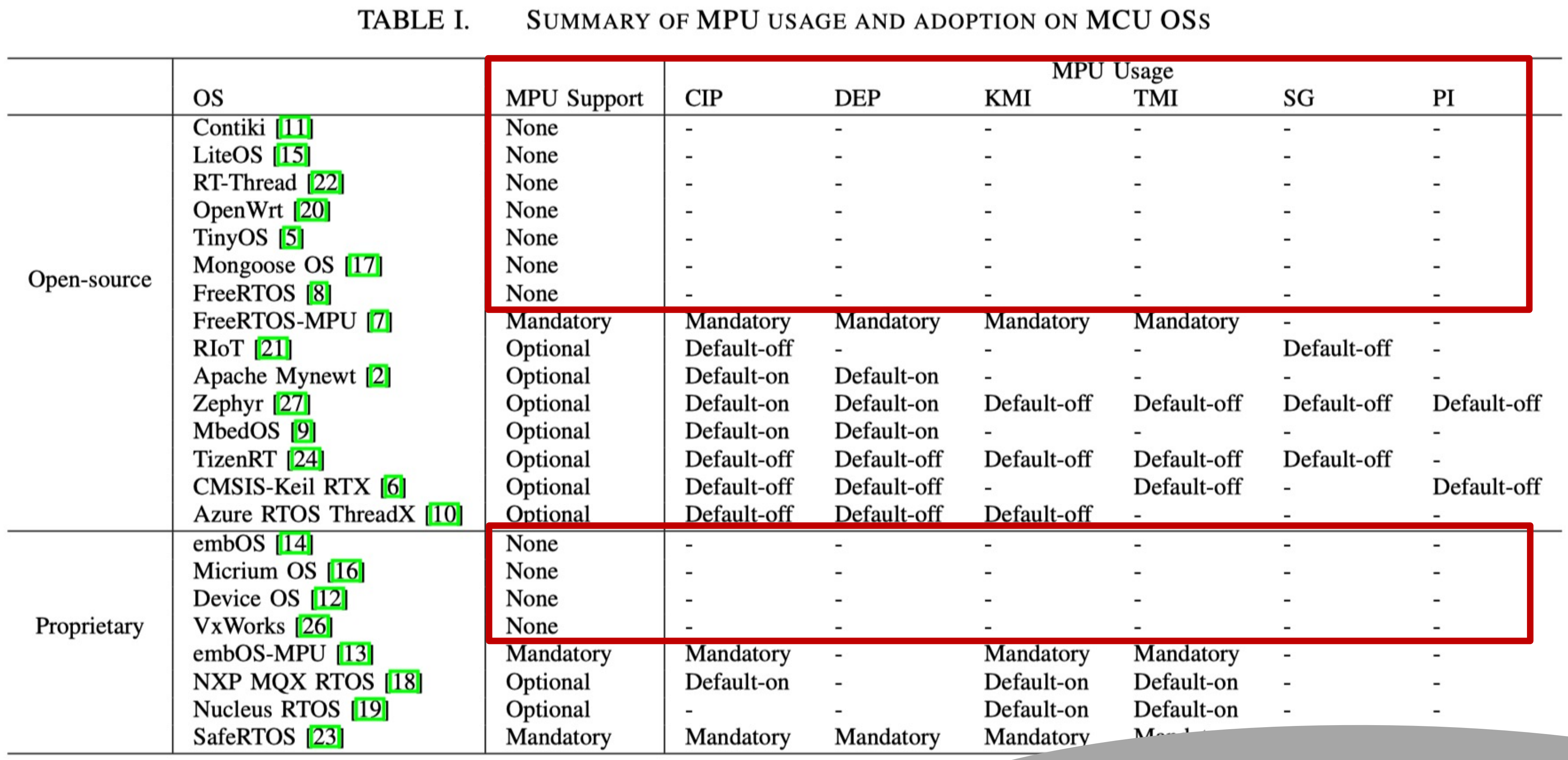


TABLE I. SUMMARY OF MPU USAGE AND ADOPTION ON MCU OSS

	OS	MPU Support	MPU Usage					
			CIP	DEP	KMI	TMI	SG	PI
Open-source	Contiki [11]	None	-	-	-	-	-	-
	LiteOS [15]	None	-	-	-	-	-	-
	RT-Thread [22]	None	-	-	-	-	-	-
	OpenWrt [20]	None	-	-	-	-	-	-
	TinyOS [5]	None	-	-	-	-	-	-
	Mongoose OS [17]	None	-	-	-	-	-	-
	FreeRTOS [8]	None	-	-	-	-	-	-
	FreeRTOS-MPU [7]	Mandatory	Mandatory	Mandatory	Mandatory	Mandatory	-	-
	RIoT [21]	Optional	Default-off	-	-	-	Default-off	-
	Apache Mynewt [2]	Optional	Default-on	Default-on	-	-	-	-
	Zephyr [27]	Optional	Default-on	Default-on	Default-off	Default-off	Default-off	Default-off
	MbedOS [9]	Optional	Default-on	Default-on	-	-	-	-
	TizenRT [24]	Optional	Default-off	Default-off	Default-off	Default-off	Default-off	-
	CMSIS-Keil RTX [6]	Optional	Default-off	Default-off	-	Default-off	-	Default-off
	Azure RTOS ThreadX [10]	Optional	Default-off	Default-off	Default-off	-	-	-
Proprietary	embOS [14]	None	-	-	-	-	-	-
	Micrium OS [16]	None	-	-	-	-	-	-
	Device OS [12]	None	-	-	-	-	-	-
	VxWorks [26]	None	-	-	-	-	-	-
	embOS-MPU [13]	Mandatory	Mandatory	-	Mandatory	Mandatory	-	-
	NXP MQX RTOS [18]	Optional	Default-on	-	Default-on	Default-on	-	-
	Nucleus RTOS [19]	Optional	-	-	Default-on	Default-on	-	-
	SafeRTOS [23]	Mandatory	Mandatory	Mandatory	Mandatory	Mand		

- Only a few MCU OSs use MPU, especially
- Even if MPU is supported, only a few security

We try to find out the reasons in this work.

Research Question

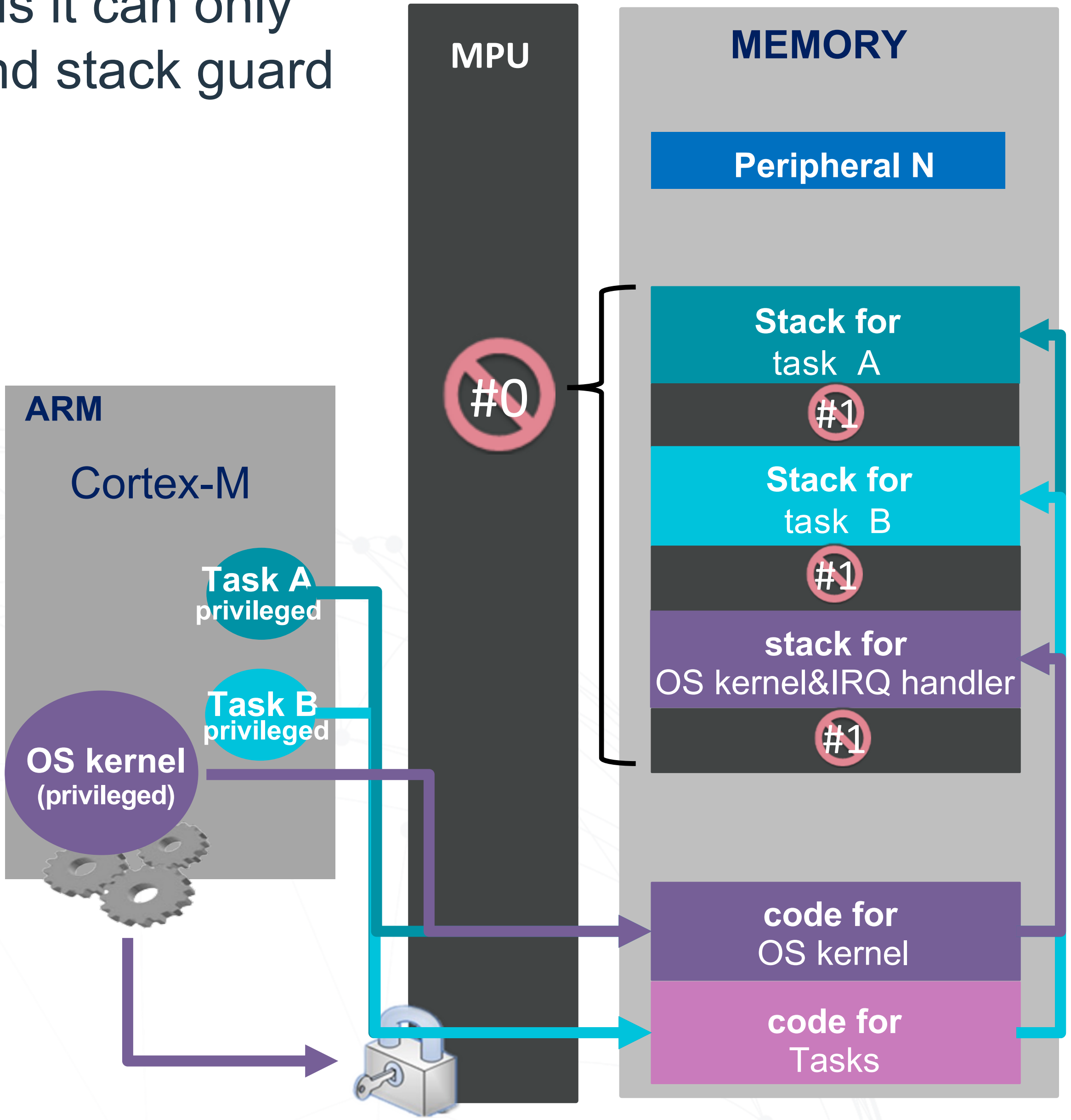
- *Why haven't MPUs been widely used in IoT operating systems?*
- *Why do only a few OSs enable full-blown MPU-enable protection by default?*
- *How do the OSs implement MPU-enable protections in detail?*
- *Are MPU-enable protections effective?*

Case Study： MPU-enabled RIoT

- RIoT runs all the code under privileged level, thus it can only provide some basic protections such as DEP, and stack guard (SG) with MPU

- **Data Execution Prevention (DEP):** RIoT enables the MPU region number 0 to cover the whole RAM region as non-executable

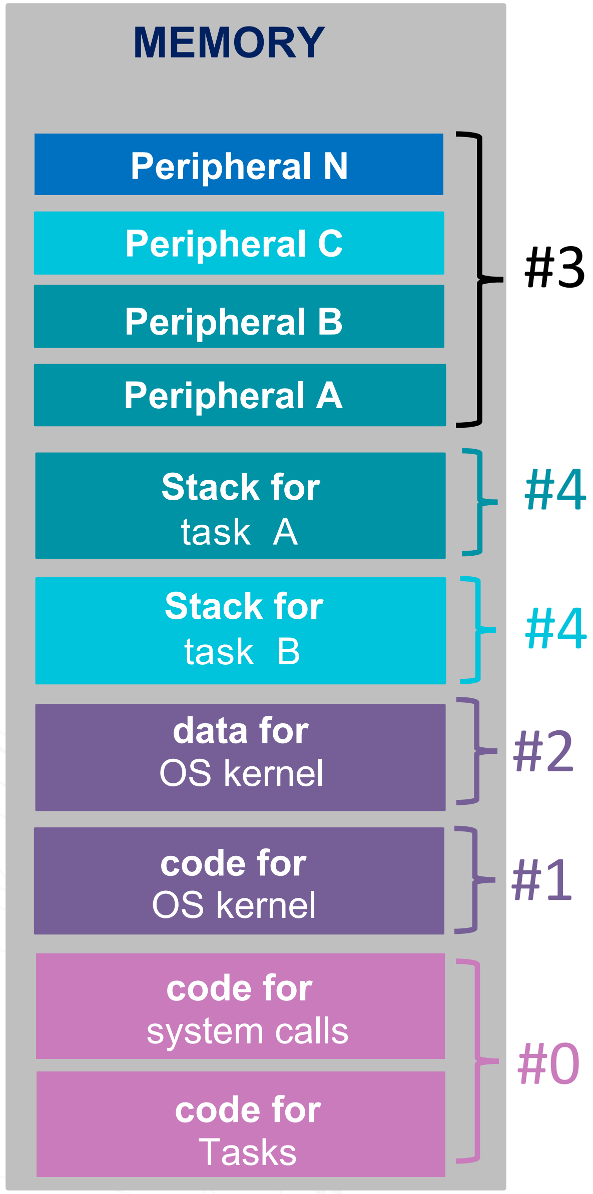
- **Stack Guard (SG):** RIoT sets last 32 bytes (the smallest MPU region) on the main stack as **read-only** via the MPU region number 1. Similarly, when switching to another task, RIoT configures the last 32 bytes of the target task stack as read-only.



Case Study： FreeRTOS-MPU

Region No.	Range	Usage	Privilege Mode	Permission
0	flash_segement_start -flash_segemnt_end	Non-writable Code Segment	Privileged Unprivileged	`r-x` `r-x`
1	privileged_functions_start -privileged_functions_end	Kernel APIs Isolation	Privileged Unprivileged	`r-x` ∅
2	privileged_data_start -privileged_data_end	Kernel Data Isolation	Privileged Unprivileged	`rw-` ∅
3	0x40000000-0x5fffffff	Non-executable Peripherals	Privileged Unprivileged	`rw-` `rw-`
4	User Task Stack	User Task Stack Isolation	Privileged Unprivileged	`rw-` `rw-`

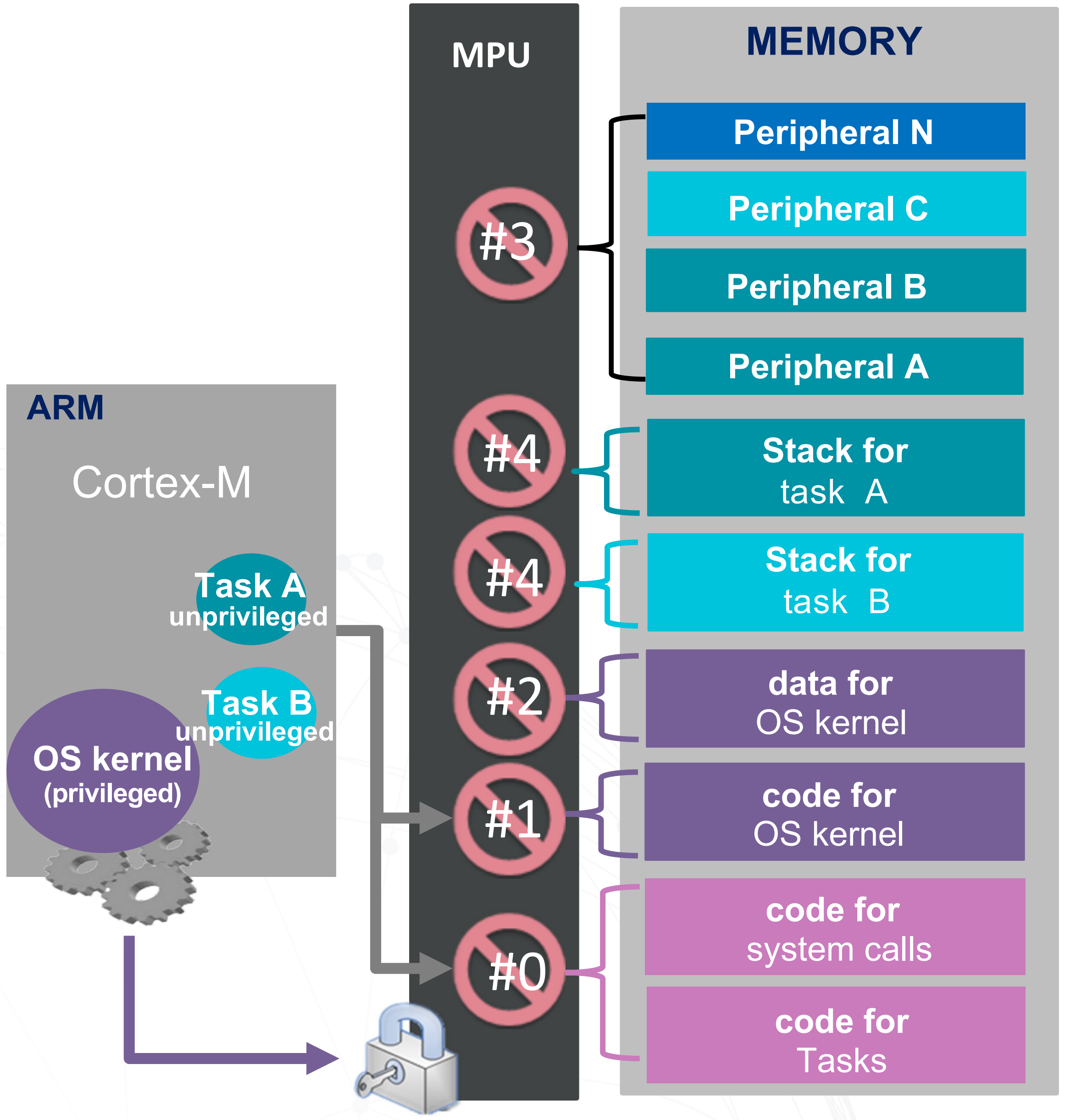
- Background region in grey is enabled for privileged access only



Case Study： FreeRTOS-MPU

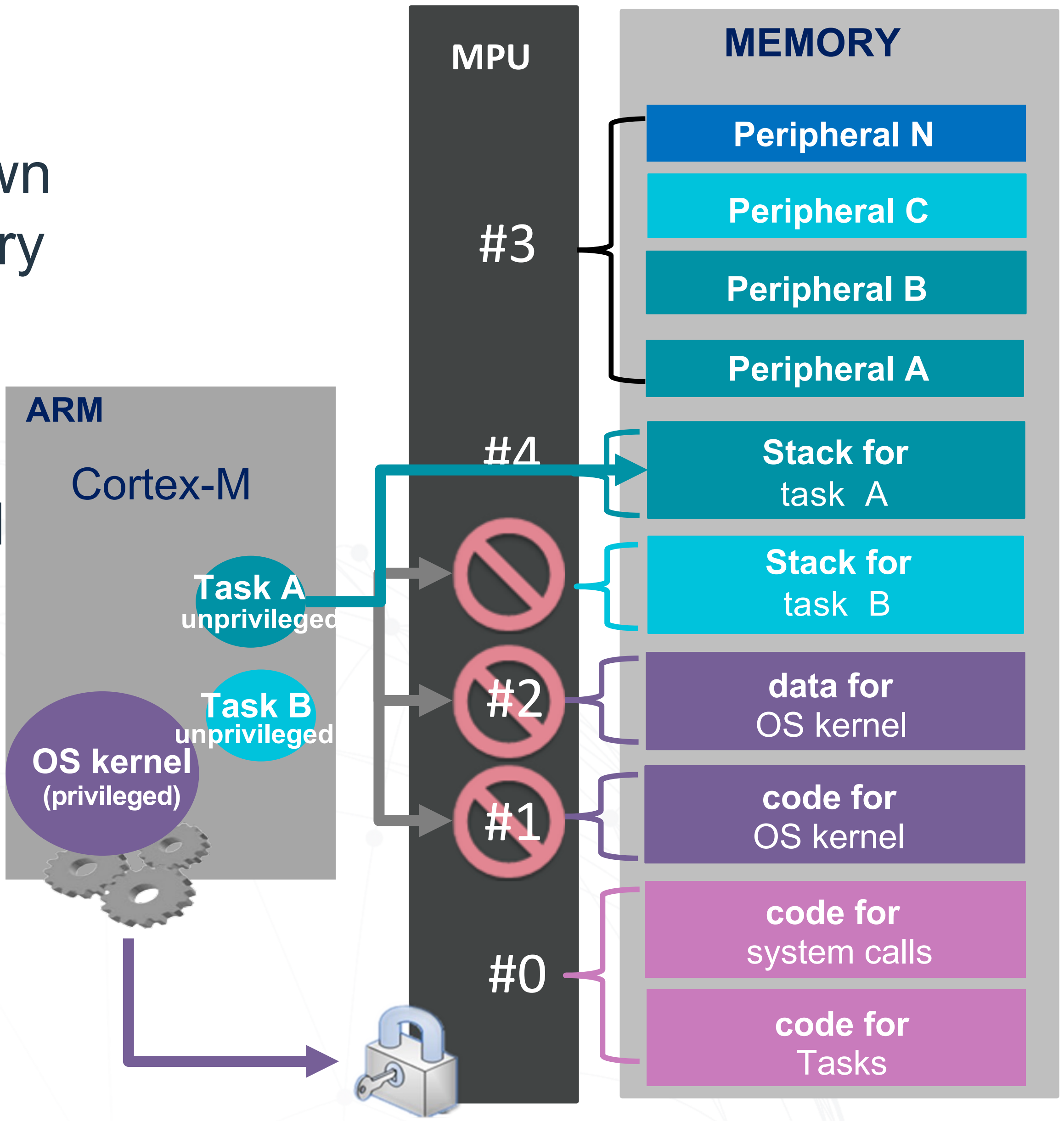
- **Code Integrity Protection (CIP):** All code regions including tasks, system calls and kernel cannot be written.

- **Data Execution Prevention (DEP):** All data regions including tasks, kernel, and peripheral regions are non-executable



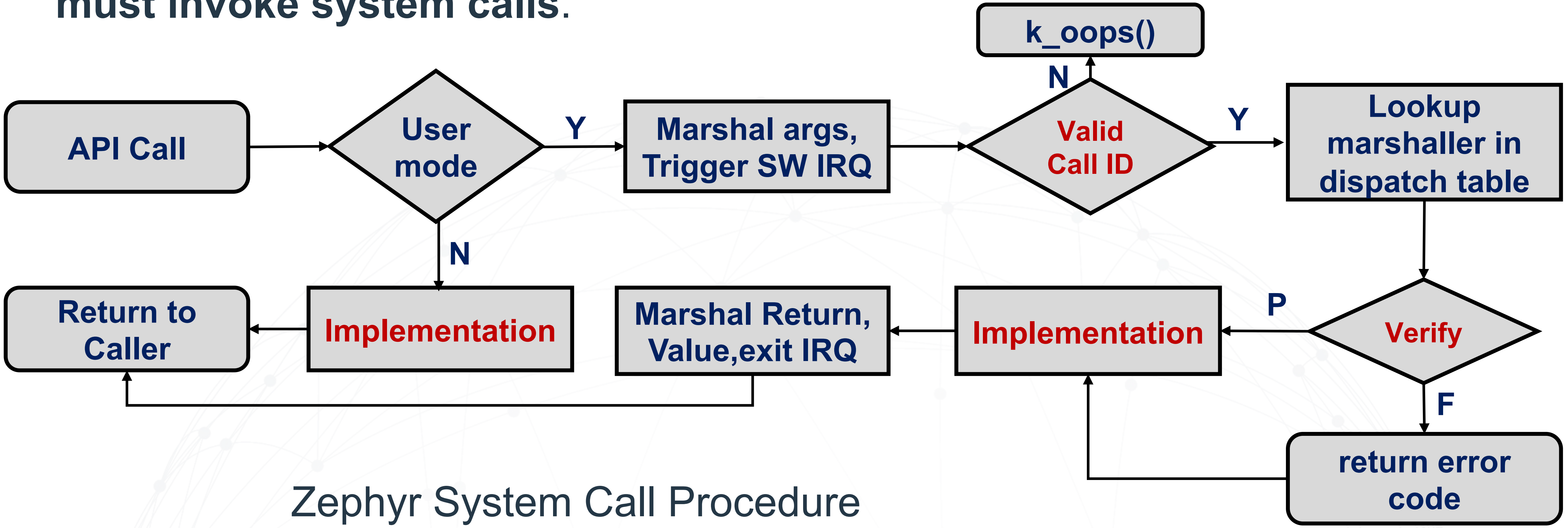
Case Study： FreeRTOS-MPU

- **User Task Memory Isolation (TMI):** Unprivileged tasks can only access their own stack and up to three user definable memory regions (three per task)
- **Kernel Memory Isolation (KMI):** The FreeRTOS kernel API and data are located in a region of Flash that can only be accessed while the microcontroller is in privileged mode (calling a system call causes a temporary switch to privileged mode)



Case Study : Zephyr

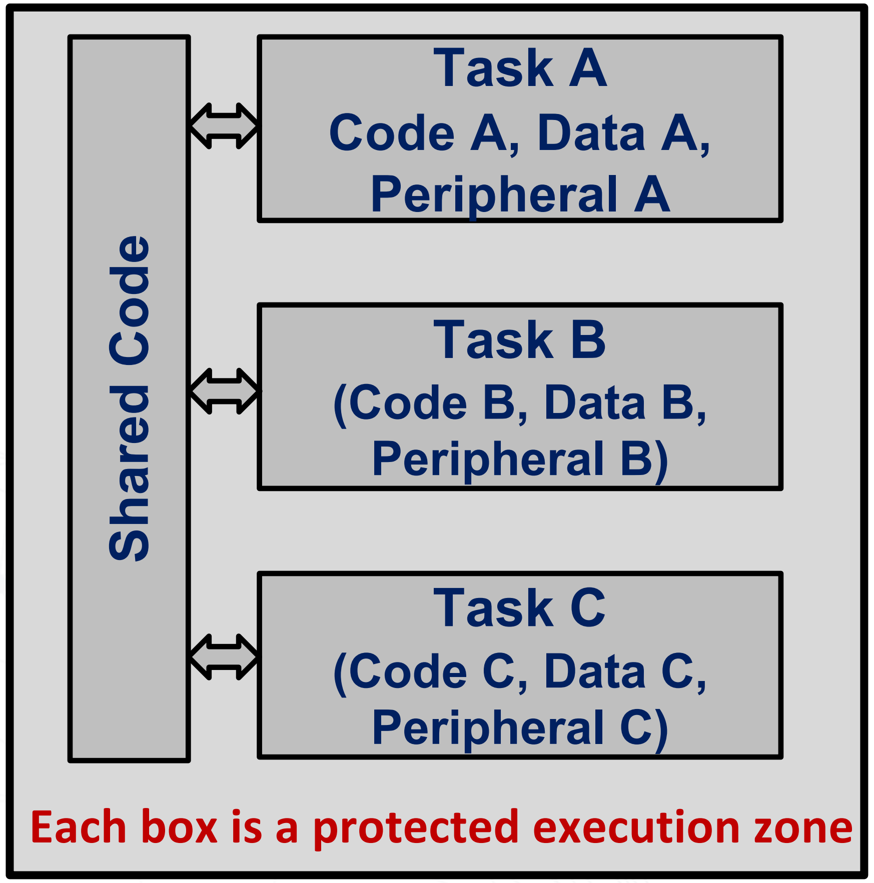
- **Peripheral Isolation (PI):** By default, a peripheral driver instance is considered as a kernel object. Therefore, its range is pre-configured to be inaccessible by user tasks. **To access peripherals, a user task must invoke system calls**.



Zephyr System Call Procedure

Case Study: Keil RTX CMSIS-Zone

- **TMI:** Based to user configuration, CMSIS-Zone assigns the access permission (RWX) to a bundle of memory regions and peripherals **as an isolated execution zone**.
- However, Keil RTX does not offer isolation between normal user tasks and the kernel (i.e., KMI is unsupported), so code and data of the kernel cannot be entirely hidden to user tasks.



Agenda

- Introduction to Memory Protection Unit (MPU)
- MPU adoption in the wild
- **Common pitfalls and limitations in using MPU**
- Mitigation suggestions
- Summary and disclosure

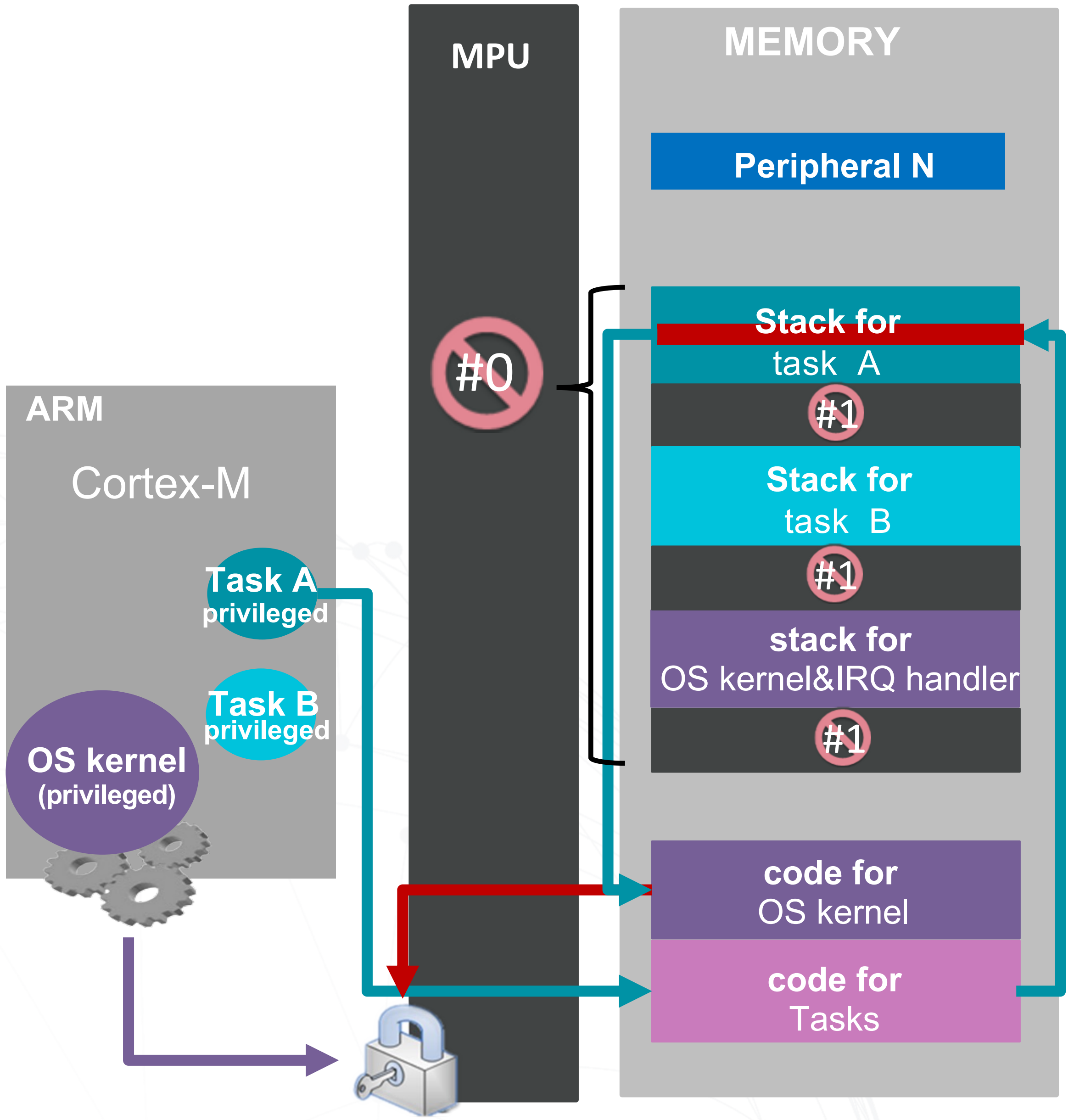
Common pitfalls in using MPU

- **Weak protection**
  - **Case study: Bypassing MPU protection in RIoT-MPU**
  - **Case study: Privileged escalation in FreeRTOS-MPU**
- Incomplete protection
- Prohibitive overhead
- Conflict with existing system designs
- Fragmented programming Interface

Bypassing MPU in MPU-enabled RIoT

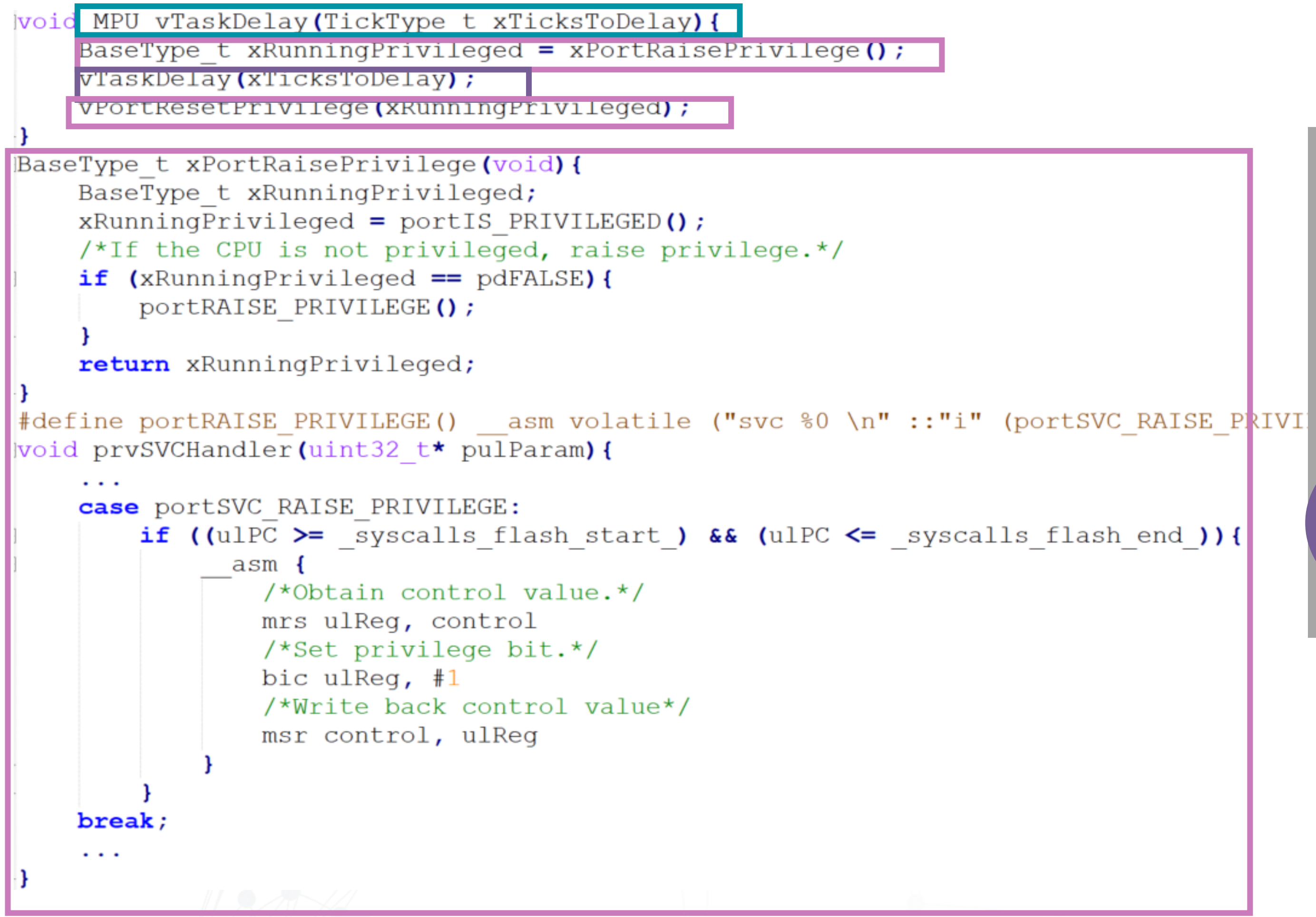
- Bug: MPU can be disabled by control flow hijacking attack (e.g., ROP)
- Cause: MPU control registers (e.g., MPU_CTRL) are located in the system peripheral region, which can be accessed by any privileged code. RIoT also provides an easy-to-use driver APIs for MPU configurations (e.g., mpu_enable and mpu_disable driver APIs).

```
int mpu_disable(void) {
#if __MPU_PRESENT
    MPU->CTRL &= ~MPU_CTRL_ENABLE_Msk;
    return 0;
#else
    return -1;
#endif
}
```

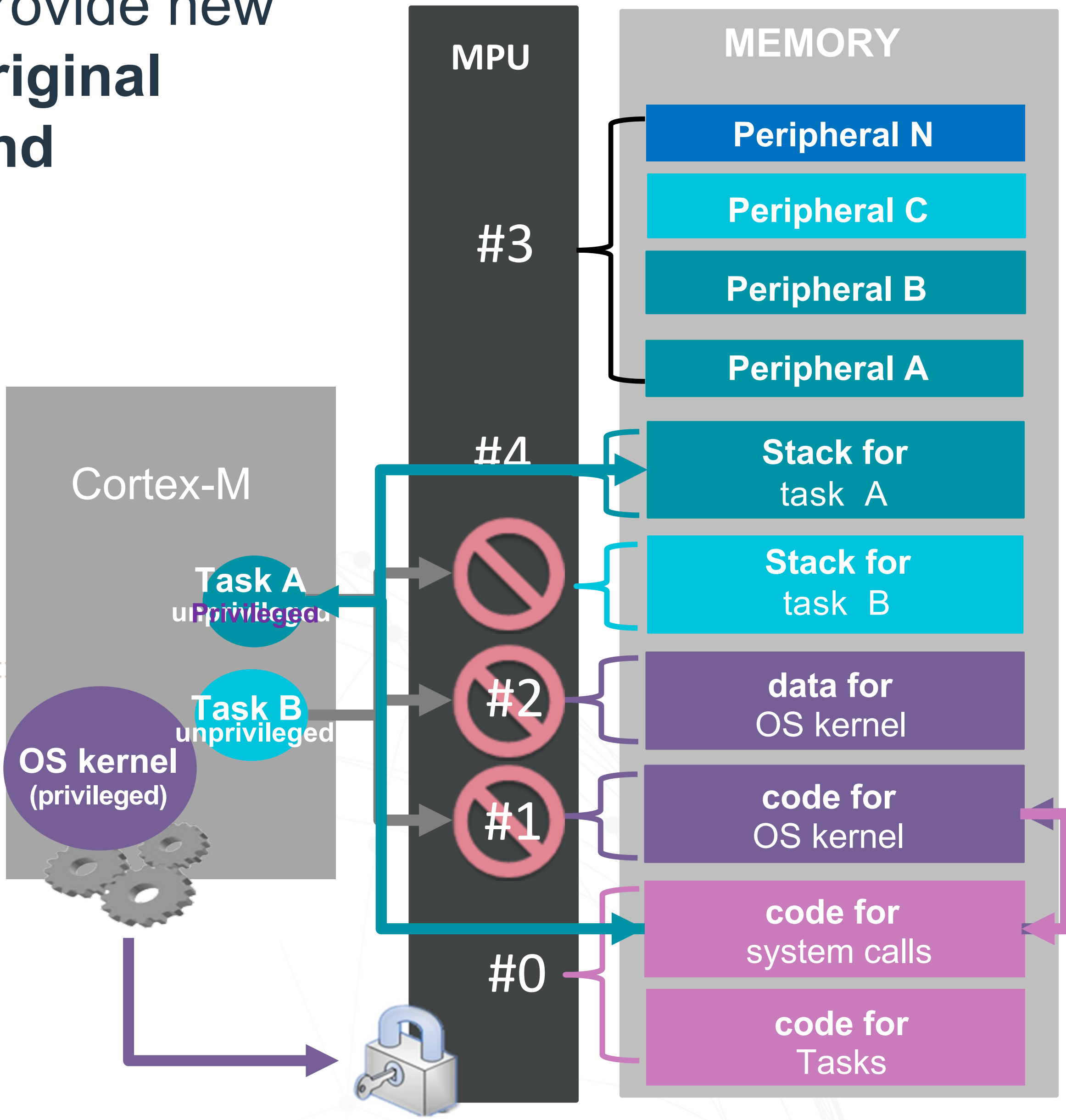


System call in FreeRTOS-MPU

- For compatibility, FreeRTOS MPU does not provide new kernel APIs for system calls, but **wraps the original kernel APIs with the xPortRaisePrivilege and vPortResetPrivilege** to raise/drop privileges



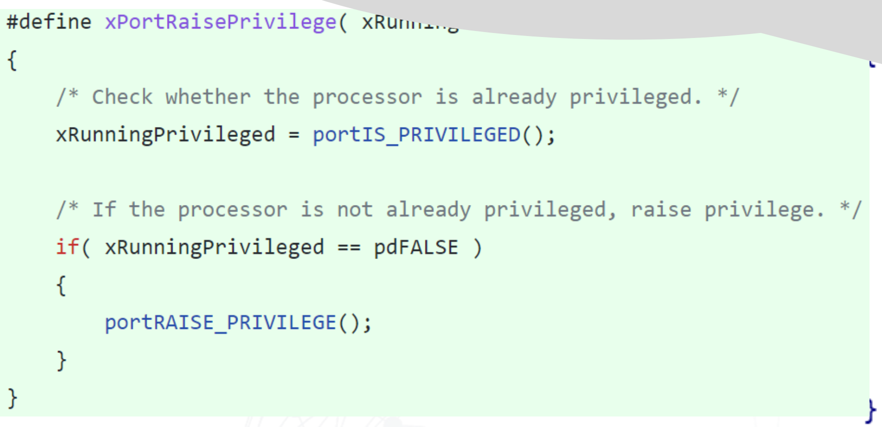
```
void MPU_vTaskDelay(TickType_t xTicksToDelay){
    BaseType_t xRunningPrivileged = xPortRaisePrivilege();
    vTaskDelay(xTicksToDelay);
    vPortResetPrivilege(xRunningPrivileged);
}
BaseType_t xPortRaisePrivilege(void){
    BaseType_t xRunningPrivileged;
    xRunningPrivileged = portIS_PRIVILEGED();
    /*If the CPU is not privileged, raise privilege.*/
    if (xRunningPrivileged == pdFALSE){
        portRAISE_PRIVILEGE();
    }
    return xRunningPrivileged;
}
#define portRAISE_PRIVILEGE() __asm volatile ("svc %0 \n" ::"i" (portSVC_RAISE_PRIVI
void prvSVCHandler(uint32_t* pulParam){
    ...
    case portSVC_RAISE_PRIVILEGE:
        if ((ulPC >= __syscalls_flash_start__) && (ulPC <= __syscalls_flash_end__)){
            __asm {
                /*Obtain control value.*/
                mrs ulReg, control
                /*Set privilege bit.*/
                bic ulReg, #1
                /*Write back control value*/
                msr control, ulReg
            }
        }
    break;
    ...
}
```



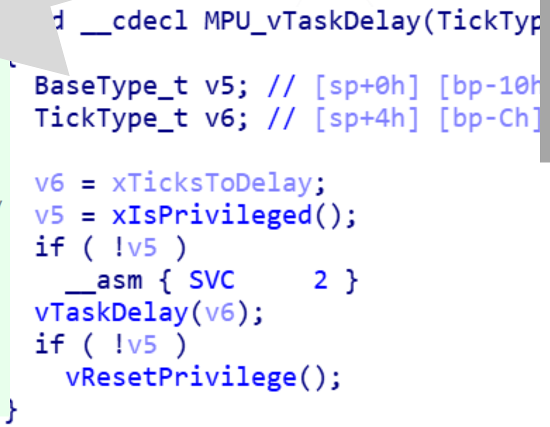
Privilege escalation in FreeRTOS-MPU

- Bug1 (v10.4.5 and before): An unprivileged task can raise its privilege by calling the internal function xPortRaisePrivilege
- Cause: Privilege escalation function (xPortRaisePrivilege) is a kernel function which can be called directly
- Patch (v10.4.6): Change it to macro implementation

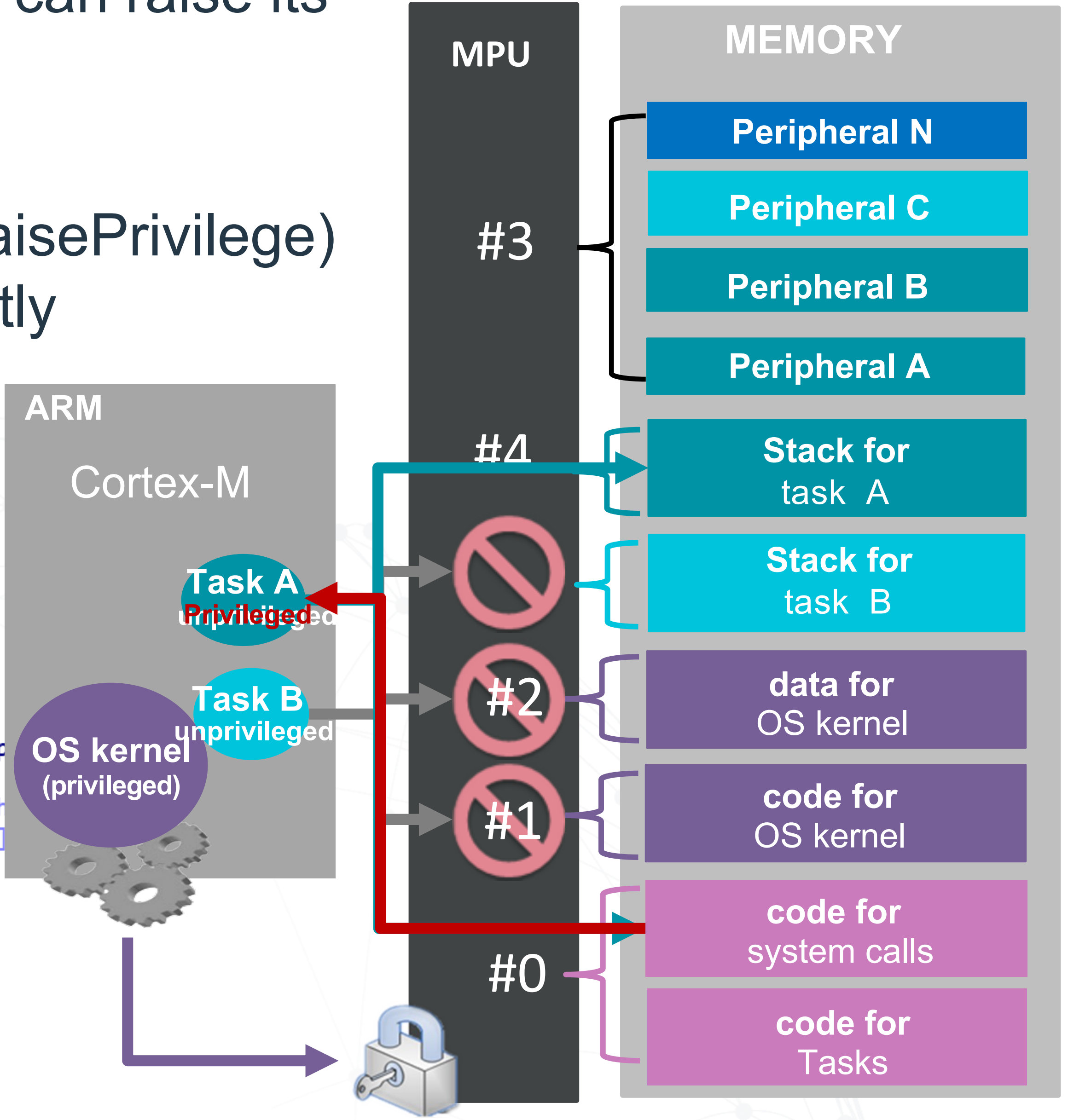
Is problem solved?



```
#define xPortRaisePrivilege( xRunning
{
    /* Check whether the processor is already privileged. */
    xRunningPrivileged = portIS_PRIVILEGED();

    /* If the processor is not already privileged, raise privilege. */
    if( xRunningPrivileged == pdFALSE )
    {
        portRAISE_PRIVILEGE();
    }
}
```



```
d __cdecl MPU_vTaskDelay(TickTyp

BaseType_t v5; // [sp+0h] [bp-10h
TickType_t v6; // [sp+4h] [bp-Ch]

v6 = xTicksToDelay;
v5 = xIsPrivileged();
if ( !v5 )
  __asm { SVC     2 }
vTaskDelay(v6);
if ( !v5 )
  vResetPrivilege();
}
```

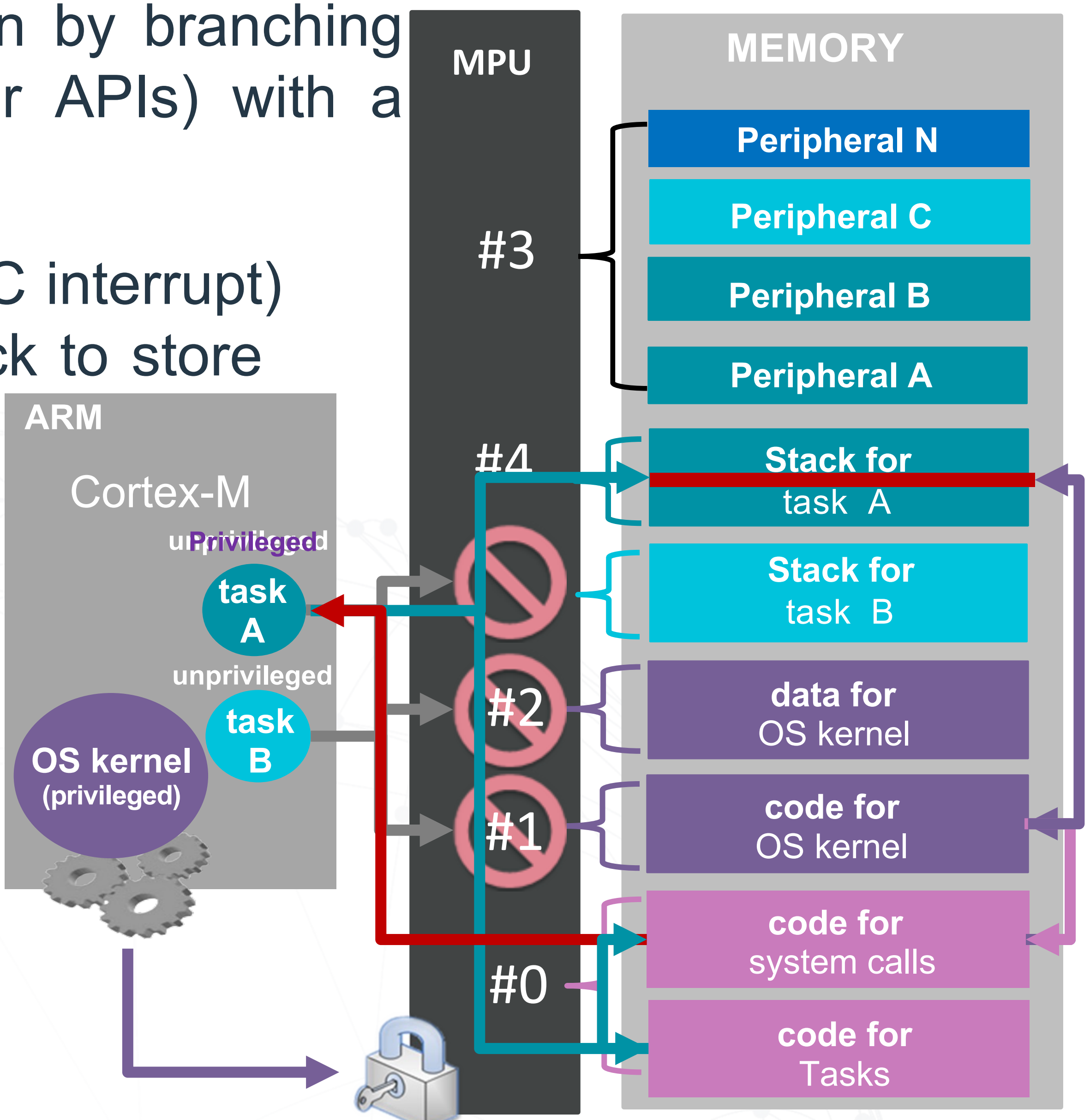


Privilege escalation in FreeRTOS-MPU

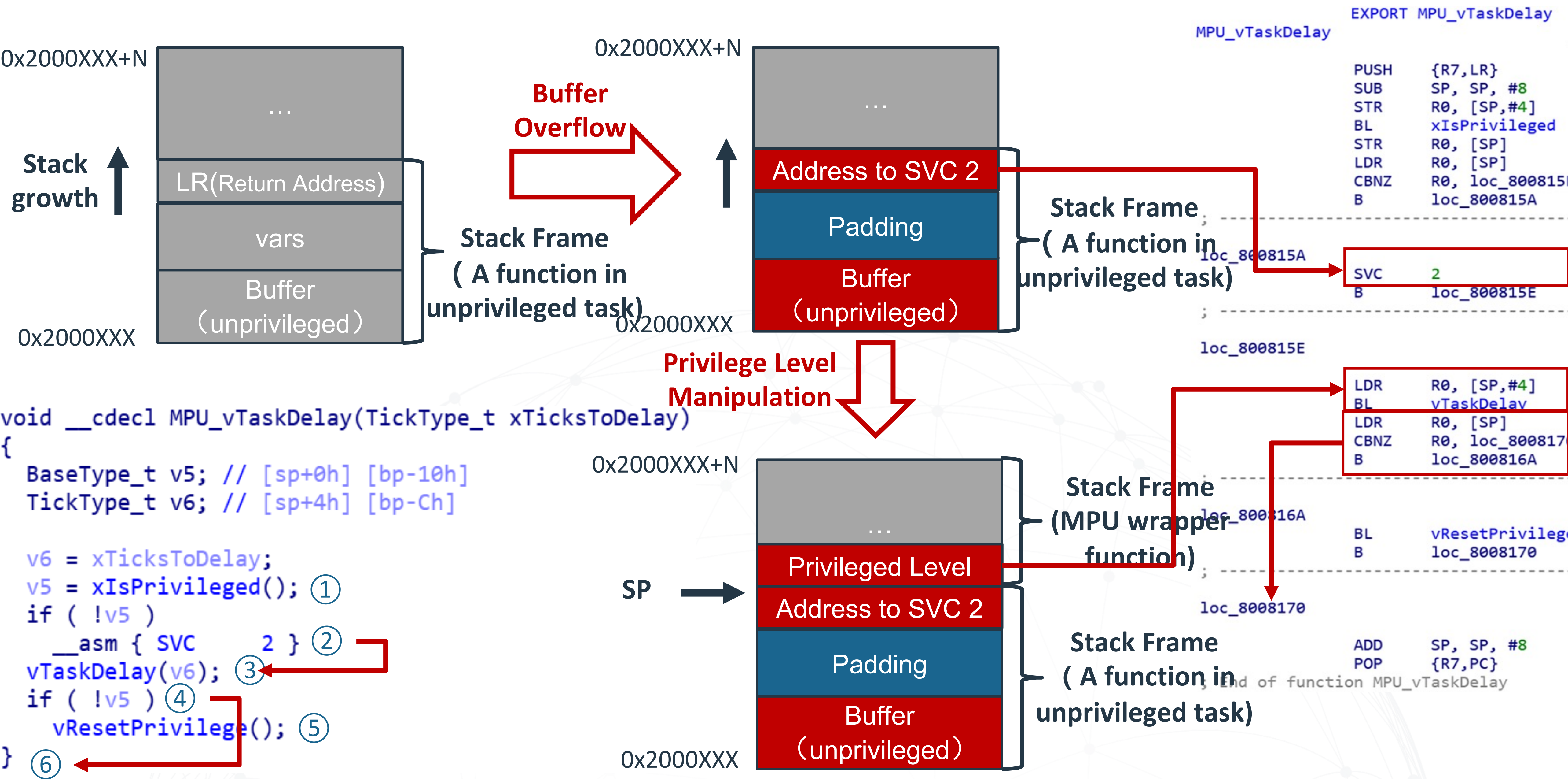
- Bug2 (v10.4.6 and before): Privilege escalation by branching directly inside system calls (MPU wrapper APIs) with a manually crafted stack frame

- Causes: Privilege escalation operation (SVC interrupt) is separated with kernel API and uses stack to store the original privilege level

```
void __cdecl MPU_vTaskDelay(TickType_t xTicksToDelay)
{
  BaseType_t v5; // [sp+0h] [bp-10h]
  TickType_t v6; // [sp+4h] [bp-Ch]

  v6 = xTicksToDelay;
  v5 = xIsPrivileged(); ①
  if ( !v5 )
    __asm { SVC      2 } ②
  vTaskDelay(v6); ③
  if ( !v5 ) ④
    vResetPrivilege(); ⑤
} ⑥
```



Exploitation Steps



```
void __cdecl MPU_vTaskDelay(TickType_t xTicksToDelay)
{
  BaseType_t v5; // [sp+0h] [bp-10h]
  TickType_t v6; // [sp+4h] [bp-Ch]

  v6 = xTicksToDelay;
  v5 = xIsPrivileged(); ①
  if ( !v5 )
    __asm { SVC     2 } ②
  vTaskDelay(v6); ③
  if ( !v5 ) ④
    vResetPrivilege(); ⑤
} ⑥
```

```
                 EXPORT MPU_vTaskDelay
MPU_vTaskDelay

                 PUSH     {R7,LR}
                 SUB      SP, SP, #8
                 STR      R0, [SP,#4]
                 BL       xIsPrivileged
                 STR      R0, [SP]
                 LDR      R0, [SP]
                 CBNZ     R0, loc_800815
                 B        loc_800815A
; ------------------------------------------

loc_800815A
                 SVC      2
                 B        loc_800815E
; ------------------------------------------

loc_800815E
                 LDR      R0, [SP,#4]
                 BL       vTaskDelay
                 LDR      R0, [SP]
                 CBNZ     R0, loc_800817
                 B        loc_800816A
; ------------------------------------------

loc_800816A
                 BL       vResetPrivileg
                 B        loc_8008170
; ------------------------------------------

loc_8008170
                 ADD      SP, SP, #8
                 POP      {R7,PC}
; End of function MPU_vTaskDelay
```

Patch

- Decide the original privilege level at the beginning using control register
- Introduced the portMEMORY_BARRIER macro to prevent instruction re-ordering when GCC link time optimization is used

```
void MPU_vTaskDelay( TickType_t xTicksToDelay ) /* FREERTOS_SYSTEM_CALL */ {
    BaseType_t xRunningPrivileged;
    if( portIS_PRIVILEGED() == pdFALSE ) {
        portRAISE_PRIVILEGE();
        portMEMORY_BARRIER();

        xPortRaisePrivilege( xRunningPrivileged );
        vTaskDelay( xTicksToDelay );
        vPortResetPrivilege( xRunningPrivileged );
        portMEMORY_BARRIER();

        portRESET_PRIVILEGE();
        portMEMORY_BARRIER();
    }
    else
    {
        vTaskDelay( xTicksToDelay );
    }
 }
```

Common pitfalls in using MPU

- Weak protection
  - Case study: Bypassing MPU protection in RIoT-MPU
  - Case study: Privileged escalation in FreeRTOS-MPU
- **Incomplete protection**
- **Prohibitive overhead**
- **Conflict with existing system designs**
- **Fragmented Programming Interface**

Incomplete protection

- Unsafe interrupt handlers
  - Exception vector reads from the Vector Address Table always use the default system address map and are not subject to an MPU check
  - Interrupt handlers (handle mode) run in the privileged mode, which can access any resources
- Incomplete protection for peripherals
  - Any load, store or instruction fetch transactions to the PPB, within the range 0xE0000000-0xE00FFFFF (system peripherals), are not subject to an MPU check.
  - Due to the programming constrains (e.g., 32B granules and alignment requirement), MPU is not suitable for protecting peripherals with small regions

Incomplete protection

- Incomplete permissions assignment
  - No execute-only (XO) permission
  - Privileged permission ≥ Unprivileged permissions
- Arm MPU does not restrict peripherals as master, allowing them to access all memory (e.g., via DMA)

Prohibitive overhead

- To leverage MPU to realize kernel/task isolation, invocation to kernel APIs has to go through context switch twice
  - Our experiment shows that one thousand privilege switches in a FreeRTOS-MPU system **takes 3.5ms** on average on the MPS2+ FPGA prototyping system broad (Cortex-M4 AN386) with 25MHZ CPU clock frequency.
- MPU regions need to be re-configured for different tasks and applications.
  - FreeRTOS has to reset MPU regions #5-7 during an application switch
  - Tizen has to reset MPU regions #3-7 during an app (including multiple tasks) switch and #6 and #7 during a task switch

Conflict with exiting system design

- Limited MPU regions for real world applications
  - Very few available user-defined regions for peripheral isolation
    - No OS provides peripheral isolation by default.
  - Very few available regions shared between two tasks
    - No OS provides shared memory protection by default.
  - It is impossible to enable too many security features at same times
    - E.g., activating all MPU features in Tizen exhausts all MPU regions
- Porting software leveraging MPU may cause compatibility issues
  - Only 30% manufacturers implement MCU hardware security features in current designs

Fragmented Programming Interface

- Chip vendors may design customized MPUs, which provide better security guarantees, but impose a steep learning curve for developers
- This may discourage them from adopting MPUs or even lead to programming errors

- Case study: NXP‘s Kinetis series MCUs discard ARM MPU and integrate NXP’s proprietary system MPU (i.e., sysMPU)
- It can restrict the permission of peripherals as masters
- When enabled, by default, peripherals cannot access RAM
- To use it, developers have to properly configure sysMPU. Otherwise, many official demos cannot execute

Agenda

- Introduction to Memory Protection Unit (MPU)
- MPU adoption in the wild
- Common pitfalls and limitations in using MPU
- **Mitigation suggestions**
- Summary and disclosure

Minimizing pitfalls

- Be careful about permission overlap
  - **Observation:** all open-source OSs but the latest FreeRTOSv10.5 use lower-number MPU regions for kernel protections ().
  - **Risk:** Developer could configure those higher numbered user-defined MPU regions to override kernel protections.
  - **Recommendation:** System and general protection (e.g., KMI, DEP,CIP) should use higher-number MPU regions.

Minimizing pitfalls

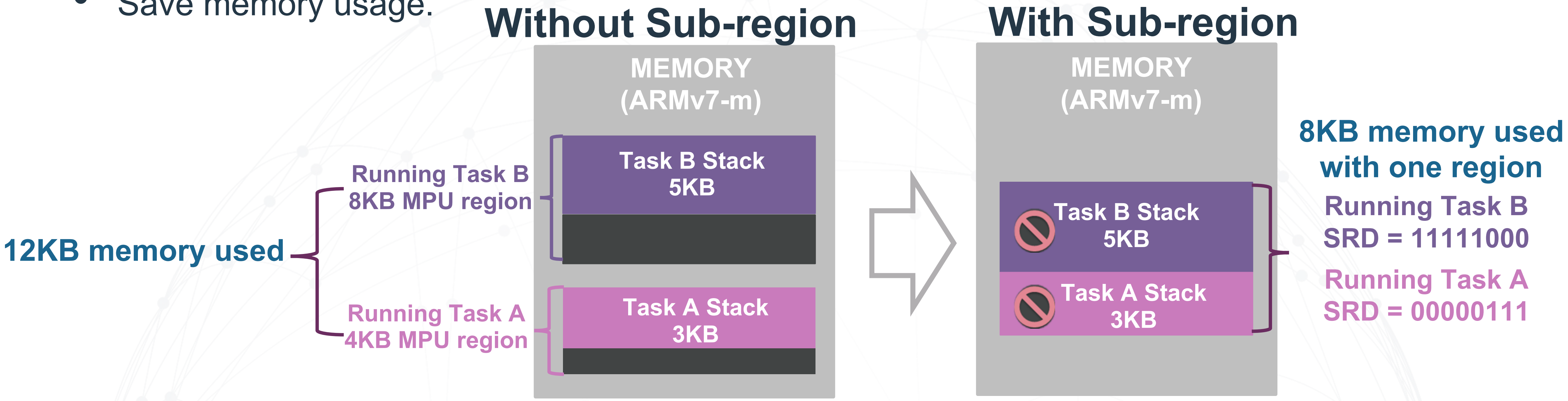
- Be careful about privilege switch during system calls
  - **Observation:** OSs wrap the kernel APIs by temperately raising and dropping privilege (e.g., FreeRTOSv10.5.0 before).
  - **Risk:** Privilege escalation with control flow hijacking or a manipulable stack.
  - **Recommendation:** MCU OSs should provide a system call interface with software interrupts, similar to traditional OSs. Alternatively, they can enforce additional caller checks before system call invocations, and the kernel should make sure that the privilege is dropped after system calls.

Minimizing pitfalls

- Avoid over-privileged execution
  - **Observation:** OSs which only provide protections like Stack Guard, DEP and CIP, always run the whole system at the privileged level like RIoT.
  - **Risk:** Disabling the desired protections by reconfiguring MPUs with control flow hijacking attack.
  - **Recommendation:** System drop privilege whenever possible.

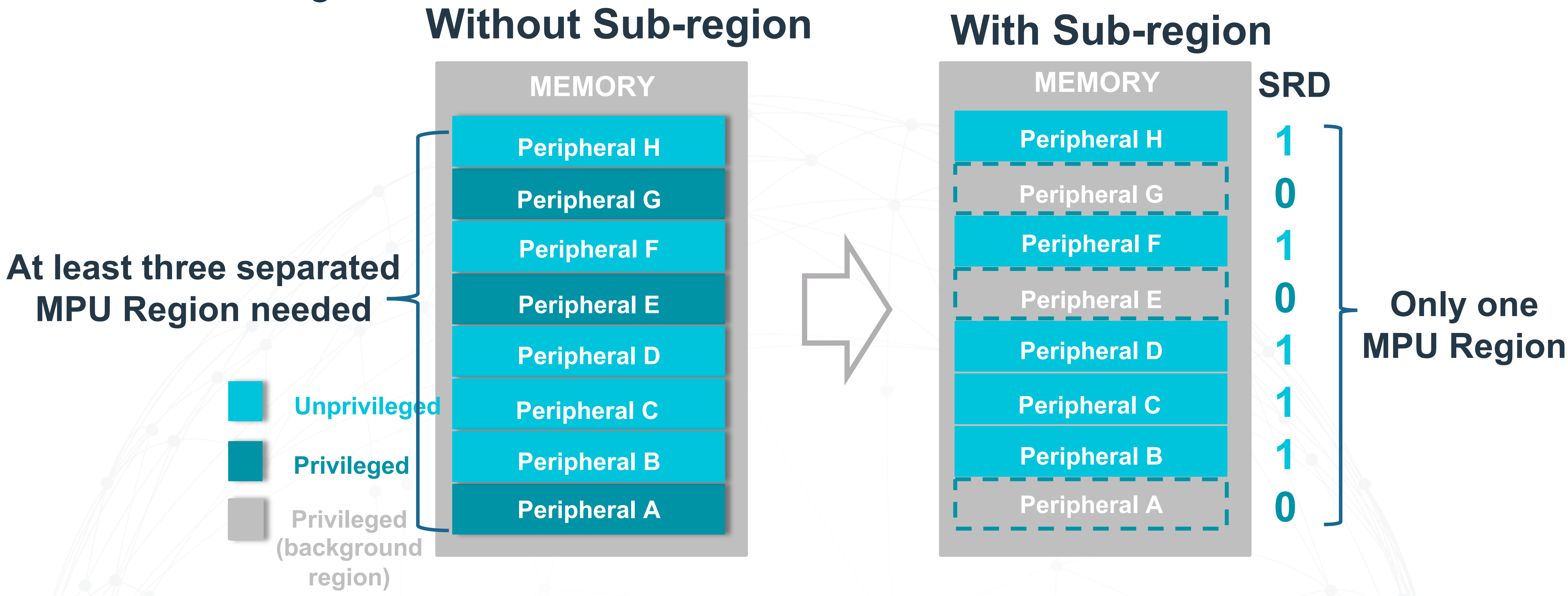
Region usage optimization

- Leverage the default MPU region on ARMv7-M
  - Default memory access permissions/attributes of memory regions is enforced by ARM without MPU
  - E.g., non-executable for standard and system peripheral regions
- Leverage sub-regions
  - Save memory usage.


Without Sub-region
With Sub-region
MEMORY (ARMv7-m)
Task B Stack 5KB
Task A Stack 3KB
Running Task B 8KB MPU region
Running Task A 4KB MPU region
12KB memory used
MEMORY (ARMv7-m)
Task B Stack 5KB
Task A Stack 3KB
8KB memory used with one region
Running Task B SRD = 11111000
Running Task A SRD = 00000111

Region Usage Optimization

- Leverage sub-regions
  - Save memory usage
  - Save MPU regions



Region Usage Optimization

- Automatically find optimized region allocation strategies
  - MINION (NDSS 2018)
  - ACES (USENIX Security 2018)

New Hardware Security Features

- ARMv8-M architecture extends the TrustZone technology to Cortex-M series. The secure regions can be used as additional regions and be assigned with higher privileged level beyond privileged level in normal world.
- Trustlite proposed **execution-aware MPU** which the not only validates data accesses (read/write/execute) but additionally considers the currently active instruction pointer as the subject performing the access.

Agenda

- Introduction to Memory Protection Unit (MPU)
- MPU adoption in the wild
- Common pitfalls and limitations in using MPU
- Mitigation suggestions
- **Summary and disclosure**

Summary

- To our surprise, we found **that MPU as a ready-to-use security feature** for protecting microcontroller **is rarely used in real-world products**
- We studied the source code of multiple MCU OSs to find explanations for this situation and eventually **identified some common pitfalls**.
- Some of the flaws **are fundamental and not remedial in a short term**
- We give **recommendations for better use of MPU**

Disclosure

- All bugs we demonstrated has been patched in latest FreeRTOS kernel
  - Security update Reference: https://www.freertos.org/security/security_updates.html
- RIoT developer team has acknowledged our finding, but the benefit of disabling access to the MPU or the `mpu_disable()` function without a userspace / kernelspace split is quite limited, only mildly increases the attack surface in the context of the attack model RIOT assumes.

Q&A

華中科技大学

UNIVERSITY OF GEORGIA